第8章 网 络

Linux几乎可以说是网络的一个同义词，事实上， Linux是一个Internet(因特网)或World Wide Web(万维网)的产品，其开发者和用户通过网络交换一些有用的思想和代码， Linux本身也经常用于网络的组织管理，本章介绍Linux是如何支持著名的TCP/IP协议族的。

TCP/IP，即传输控制协议/网际协议(Transmission Control Protocol/Internet Protocol)，实际上是一个由多种协议组成的协议族，它定义了计算机通过网络互相通信及协议族各层次之间通信的规范。

TCP/IP最初是在由美国政府资助的美国高等研究计划署的网络 ARPANET上发展起来的，该网络用于支持美国军事和计算机科学研究，正是由它提出了报文交换和网络分层概念。1988年以后， ARPANET由其继任者 —— 美国国家科学基金会的 NSFNET所取代，而NSFNET和全世纪数以万计的局域网和区域网共同连接成了一个巨大的联合体 —— 因特网(Internet)，举世闻名的万维网(World Wide Web)也是来自于ARPAnet并完全采用TCP/IP协议族。UNIX被广泛地应用于ARPANET，它的第一个网络版是4.3 BSD (Berkeley Software Distribution)，该版本支持BSD的套接字(略有扩充)和全部的TCP/IP协议，Linux的网络功能即是基于这个版本实现的。Linux之所以以该4.3 BSD版本为模型，是因为这个版本广为流行，并且它支持 Linux与其他UNIX平台之间应用程序的移植。

8.1 TCP/IP网络概述

本节将概述TCP/IP网络的主要原理。

在一个TCP/IP网络中，每台主机都分配有一个 32位的IP地址，该地址可以唯一地标识主机。IP地址通常用“ . ”隔开的四个十进制数表示，称为点分十进制表示，如 IP地址0x81124C15(16进制)通常写成129.18.76.21。

IP地址由两部分组成：网络 (network)地址和主机(host)地址。网络地址由IP地址的高位组成，主机地址由低位组成，这两部分的大小取决于网络的类型。如一个 B类地址(IP地址的第一个字节大小在128到191之间)，其IP地址的前两个字节是网络地址，后两个字节表示主机地址，这样一个B类地址可支持65536个网络，同时每个网络中可容纳65536台主机。

IP地址的主机部分可以分出多个子网，利用子网技术，大的网络 (即主机地址部分占较多字节)可以被分为若干小的子网 (subnetwork)，每一个子网均可独立维护。例如， IP地址16.42.0.9，可将其设置为子网地址16.42.0，其主机地址16.42.0.9，这种技术通常用来划分一个企业的网络，如将 16.42作为ACME计算机公司的网络地址，那么 16.42.0则为子网0，16.42.1则为子网1，这些子网或许位于相分离的建筑物中，它们通过租用的电话线甚至微波相连。通常由网络管理员为主机分配 IP地址，使用了子网技术将更有助于网络管理员的分派，并且管理员在其所辖子网内可以随意分配IP地址而不会有IP地址冲突。

一般说来，点分十进制表示的 IP地址不易记，而记名字则容易多了。因此，每台网络主机还有一个名字，由域名服务 (Damain Name Service, DNS)负责IP地址与网络主机名的互译，并在整个因特网上发布名字 —— IP地址数据库。使用网络主机名使得一台机器的 IP地址改变

(例如，这台机器被移到了另外一个网络上)时，不必担心别人在网络上找不到这台机器，这台机器的DNS记录只是更新了 IP地址，所有用网络主机名对这台机器的访问将继续有效。在Linux中，主机名字可静态地在 /etc/hosts文件中定义；也可请求分布式域名服务器 (Distributed Name Server, DNS)为其指定一个，这样主机必须应该知道一个或多个 DNS服务器的IP地址，这些信息定义在/etc/resolv.conf文件中。

当连接一台主机或访问一个Web主页时，都要通过本机的IP地址与被访问主机通信，用IP报文交换数据。IP报文分为两部分：IP报头与数据，在IP报头中，包含有源主机IP地址、目的主机IP地址、校验和和其他一些有用信息 (详见图1-8-1)，其中校验和是由源主机利用 IP报文数据计算得出的，目的主机据此判断报文在传输过程中是否被破坏。为了便于控制，应用程序传输的数据可能会被分片为更小的 IP报文，而IP报文的大小则由传输介质所决定：以太网报文通常要比点到点(Paint to Point Protocal, PPP)报文大一些，而目的主机在将数据交给应用程序之前，必须先重组报文。IP报头中的“ 标识 ”(IDENTIFICATION)域、“ 标志 ”(FLAG)域和“ 片偏移 ”(FRAGMENT OFFSET)域用来进行数据的分片与重组。如果通过较慢的网络传输图片，那么看一下图片的显示过程，即可感受到分片与重组的过程。

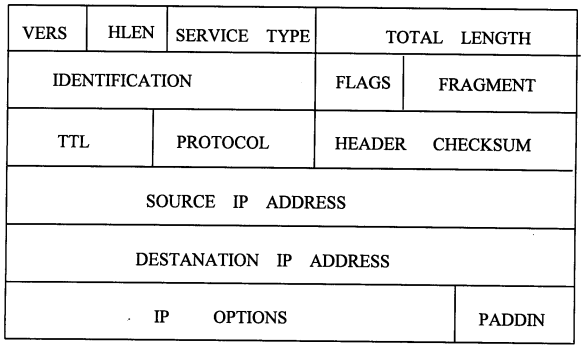

图1-8-1 IP报头数据格式

同一网络中的主机相互间可直接发送报文，但不同网络中的主机要通信，则必须通过一台特殊的主机：网关 (Gateway)。网关(或路由器)是一台同两个或更多个网络有直接连接的节点，网关可以在网络之间交换信息，把报文从一个网络传递到另一个网络。例如，网络16.42.1.0与16.42.0.0通过一个网关相连，则所有从网络 16.42.1.0发送到16.42.0.0的报文都须先发给网关，再由网关为其选择路由，转发报文。对应于每一个目的 IP地址，网关中的路由表都提供一个入口用以查询将该报文发送到哪台主机。这些路由表动态刷新，随应用程序使用网络的时机和网络技术的不同而改变。

我们可用netstat命令来查看某机器当前的路由表，其输出大致类似图 1-8-2。

```
dai %          netstat -rn
Kernel         routing        table
Destinaton     Gatevay address  Flags   Refcnt   Use      Iface
net/address
16.42.0.0      16.42.0.12       UN      0        1432     eth0
default        16.42.0.1        UGN     0        1432     eth0
17.0.0.1       17.0.0.1         UH      0        12       to
16.42.0.12     17.0.0.1         UH      0        12       to
```

图1-8-2 netstat命令输出

第一栏是路由表包括的目标网络或节点的地址。路由表第一条对应于网络 16.42.0，这是本机所在网络，任何本机发到这个网络的报文必须通过 16.42.0.12，即本机的IP地址。一般一个机器到自己网络的路由总是通过它自己。

Flags栏给出目标地址信息：U表示此路由“ up ”，N表示目标是一个网络，等等。 Refcnt和Use栏给出这条路由的使用统计。 Iface列出路由使用的网络设备。 eth0表示以太网接口， lo表示loopback设备。

路由表中第二条是缺省路由，适用于所有目的网络或带节点地址不在路由表中的报文。本例中16.42.0.1可看作通向外界的门户，所有 16.42.0子网的机器必须通过 16.42.0.1与其他网络通信。

路由表中第三条对应于地址 17.0.0.1，这是loopback地址，当机器想与自己建立 TCP/IP连接时适用这个地址，它使用 lo作为接口设备，避免了 loopback连接使用以太网接口 (eth0)，这样网络带宽不会因机器与自己对话而浪费。

路由表中最后一条是对地址 16.42.0.12，这是本机的IP地址。正如上述，它利用 17.0.0.1作为自己的网关。

连接不同网络的网关的路由表通常类似下面的例子 (图1-8-3)，假设这个网关在两个子网的地址分别为16.42.0.109和16.42.1.4。

Destination net/address	Gateway address	Flags	Refcnt	Use	Iface
16.42.0.0	16.42.0.109	UN	0	1432	eth0
16.42.1.0	16.42.1.4	UN	0	1432	eth1
default	16.42.1.43	UGN	0	1432	eth1
17.0.0.1	17.0.0.1	UH	0	12	to
16.42.0.109	17.0.0.1	UH	0	12	to

图1-8-3 netstat命令输出

本网关通过eth0设备与16.42.0网络相连，通过 eth1设备与16.42.1网络相连。如果16.42.0网络的机器想同外界通信，它将把报文发往它的网关 16.42.0.109，16.42.0.109将再把报文发往它的缺省路由，即网关 16.42.1.43。如此下去，报文从一个网关传递到下一个网关，直到到达目的网络。

从协议分层来看， IP是网络层协议， TCP是一个可靠的端到端传输层协议，它利用 IP层传接报文。TCP是面向连接的，它通过建立一条虚电路在不同的网络间传输报文，可以保证所传输报文的无丢失性和无重复性。用户数据报协议 (User Datagram Protocol, UDP)也要利用IP层传输报文，但它是一个非面向连接的传输层协议。利用 IP层传输报文时，当目的方 IP层收到IP报文后，必须能够识别出该报文所使用的上层协议 (即传输层协议)。因此，在IP报头(参见图1-8-1)中，设有一个“ 协议 ”域(PROTOCOL)。通过该域的值，即可判明其上层协议类型。传输层与网络层在功能上的最大区别是前者提供进程通信能力，而后者则不能。在进程通信的意义上，网络通信的最终地址就不仅仅是主机地址了，还包括可以描述进程的某种标识符。为此， TCP/UDP提出了协议端口(protocol port，简称端口)的概念，用于标识通信的进程。例如， Web服务器进程通常使用端口80，在/etc/services文件中有这些注册了的端口地址。

分层协议不只包括TCP、UDP与IP，IP层本身亦要用到许多不同的物理介质来传输 IP报文，这些介质也有自己的报头信息，例如以太网层。一个以太网允许多台主机同时连到一根缆线上，任一台主机发送的帧对其他所有主机都是可见的，因此每台主机都有一个唯一的以太网

地址，指明了目的以太网地址的帧将只被拥有该地址的主机接收。以太网地址是一个 48比特的整数，以机器可读的方式存入主机接口中，叫作硬件地址（ hardware address）或物理地址（physical address）。以太网地址的一个重要特性是每一地址都与一个特定主机接口联系在一起，接口与地址一一对应。以太网地址共 6字节长，为保证主机以太网地址的全球唯一性，以太网采用一种层次型地址分配方式：以太网地址管理机构 (IEEE)将以太网地址(48比特的不同组合)分成若干独立的连续地址组，生产以太网接口板的厂家从中购买一组，具体生产时，再从所购地址中逐个将唯一地址赋予接口板。以太网地址中有一些保留地址用于多目的通信，当某一以太网帧拥有这样一个目的地址时，以太网上将会有多台主机同时接收这一帧。像 IP报文一样，以太网帧同样支持多种上层协议，这样在以太网帧头中也有一个协议域，通过该域的值，以太网层即能将所收到的IP报文正确地传给IP层。

IP地址是一种简单的地址，在分配或改变 IP地址时，网络管理员可以随心所欲，但同时物理地址是固定的，而网络硬件只响应物理地址，这样就必须提供一种机制，来完成这两种地址的映射。在Linux中采用的是地址解析协议(Address Resolution Protocol, ARP)和逆向地址解析协议(Reverse Address Resolution Protocol, RARP)。ARP用于从IP地址到物理地址的映射，RARP用于从物理地址到IP地址的映射。当主机A欲解析主机B的IP地址I_B时，A首先广播一个ARP请求报文，请求IP地址为I_B的主机回答其物理地址P_B。网上所有主机(包括B)都将收到该ARP请求，但只有B识别出自己的I_B地址，并作出应答：向A发回一个ARP响应，回答自己的物理地址P_B。ARP并非只能服务于以太网，它也支持其他一些物理介质，例如 FDDI。RARP通常由网关所用，以响应对远程网络IP地址的ARP请求。

8.2 Linux中的TCP/IP网络层次结构

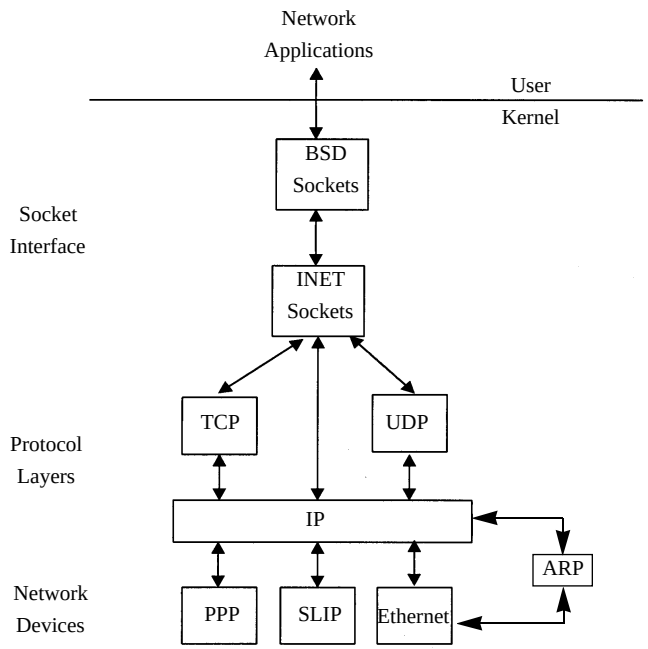

图1-8-4 Linux网络层次结构

图1-8-4描述了Linux对IP协议族的实现机制，如同网络协议自身一样， Linux也是通过视

其为一组相连的软件层来实现的。其中 BSD套接字（Socket）由通用的套接字管理软件所支持，该软件是INET套接字层，它来管理基于 IP的TCP与UDP的端到端互联问题。如前所述，TCP是一个面向连接协议，而 UDP则是一个非面向连接协议，当一个 UDP报文发送出去后，Linux并不知道也不去关心它是否成功地到达了目的主机。对于 TCP传输，传输节点间先要建立连接，然后通过该连接传输已排好序的报文，以保证传输的正确性。 IP层中代码用以实现网际协议，这些代码将IP头增加到传输数据中，同时也把收到的 IP报文正确地转送到TCP层或UDP层。IP层之下，是支持所有Linux网络应用的网络设备层，例如点到点协议 (Point to Point Protocol, PPP)和以太网层。网络设备并非总代表物理设备，其中有一些（例如回送设备）则是纯粹的软件设备，网络设备与标准的 Linux设备不同，它们不是通过 mknod命令创建的，必须是底层软件找到并进行了初始化之后，这些设备才被创建并可用。因此只有当启动了正确设置了以太网设备驱动程序的内核后，才会有 /dev/eth0文件。ARP协议位于IP层和支持地址解析的协议层之间。

8.3 BSD套接字接口

这是一个通用接口，它不仅支持不同的网络结构，同时也是一个内部进程间通信机制。一个套接字描述了一个连结的一个端点，因此两个互联的进程都要有一个描述它们之间连结的套接字，可以把套接字看作是一种特殊的管道，只是这种管道对于所包含的数据量没有限制。Linux支持套接字地址族中的多个，如下所列：

UNIX	Unix域套接字
INET	使用TCP/IP的因特网地址族
AX25	业余无线X25
IPX	IPX
APPLETALK	APPLETALK
X25	X25

Linux支持多种套接字类型。套接字类型，是指创建套接字的应用程序所希望的通信服务类型。同一协议族可能提供多种服务类型，比如 TCP/IP协议族提供的虚电路与数据报就是两种不同的通信服务类型， Linux BSD支持如下几种套接字类型：

- Stream 提供可靠的面向连接传输的数据流，保证数据传输过程中无丢失、无损坏和无冗余。INET地址族中的TCP协议支持该套接字。
- Datagram 提供数据的双向传输，但不保证消息 (message)的准确到达，即使消息能够到达，也无法保证其顺序性，并可能有冗余或损坏。 INET地址族中的UDP协议支持该套接字。
- Raw 是低于传输层的低级协议或物理网络提供的套接字类型，比如通过分析为以太网设备所创建的Raw套接字，可看到裸IP数据流。
- Reliable Delivered Messages 类似于Datagram套接字，但它可以保证数据的正确到达。
- Sequenced Packets 类似于Stream套接字，但它的报文大小是可变的。
- Packet 这是Linux对标准BSD套接字类型的扩展，它允许应用程序在设备层直接访问报文数据。

利用套接字通信的进程一般采用客户 —— 服务器模型：服务器提供服务，客户是这些服务

的使用者。例如Web服务器提供Web页，而Web用户访问这些Web页。服务器首先创建一个套接字，然后为其绑定一个名字，名字的格式取决于该套接字的地址族，由 sockaddr数据结构定义，但通常是该服务器的主机地址。一个 INET套接字还要绑定一个IP端口号，比如Web服务的端口号为80，在/etc/services文件中保存有注册过的端口号。地址绑定之后，服务器即开始在该地址上侦听连接请求，而客户端则为建立连接创建一个套接字，并指明目的地址 —— 服务器地址。对一个INET套接字而言，服务器地址就是该主机的 IP地址加上一个端口号。这些客户请求首先要能够通过多种协议层到达服务器端，然后进入等待队列，一旦服务器收到这些请求，首先要判断是否接受，若同意接受，则服务器必须再创建一个新的套接字用以接受该请求，这是因为一旦一个套接字用于侦听，那么它就不能再被用于支持连接。连接建立之后，双方即可进行数据传输；当连接不再需要时，须将其关闭。在传输过程中，要注意正确处理传输的数据报文。

一个BSD套接字操作的具体含意取决于低层的地址族，建立一个 TCP/IP连接就与建立一个业余无线X.25连接有很大不同。类似于虚文件系统， Linux利用与应用程序所用的BSD套接字接口相关的BSD套接字层将套接字接口抽象出来，同时这些应用程序所用的 BSD套接字接口又由各种独立的地址族软件所支持。初始化内核时，编入内核的各地址族将注册它们自己的BSD套接字接口；之后，当应用程序创建和使用 BSD套接字时，系统将把BSD套接字与支持该套接字的地址族联结起来，这种联结是通过交叉链接地址族例程的数据结构和表形成的。比如某一地址族定义了创建套接字例程，则当一应用程序创建一个新套接字时，将使用该例程。

当内核设置时，将会建立一些地址族和协议的协议向量 (protocol vector)，用它们的名字来代表每一个向量，例如“ INET ”和它的初始化例程地址。在系统启动时要初始化套接字接口，这时将调用每一个协议的初始化例程，这对套接字地址族而言意味着注册了一组协议操作，这些操作针对各自的地址族都做了些工作，它们被保存在 pops向量中。pops向量包括一些指向proto_ops数据结构的指针， proto_ops数据结构包括地址族类型和一组指针，这些指针指向特定地址族所定义的套接字操作例程，可利用地址族标志检索 pops向量，例如INET的标志为2。

8.4 INET的套接字层

INET套接字(socket)层支持包括TCP/IP协议在内的INET地址族(Address Family)，如前所述，这是一些分层协议，下层协议为上层协议提供服务。 Linux中实现TCP/IP协议的代码与数据结构充分体现了这种协议分层。 INET套接字层接口是通过一组INET地址族套接字操作实现的，这些操作在网络初始化时被注册到了 BSD套接字层，与其他注册的地址族一起保存在pops向量中。BSD套接字层通过调用注册在 INET proto_ops数据结构中的INET套接字层例程来完成上述操作。在进行每一项操作时， BSD套接字层都要把代表 BSD套接字的数据结构传给INET层，INET套接字层并非简单地抽取BSD套接字中的特定TCP/IP信息，而是使用自己的sock数据结构，该数据结构已被链接到 BSD socket数据结构上了，在图 1-8-5中给出了这种链接，这种链接通过BSD socket中的data(数据)指针将sock数据结构链到了BSD socket数据结构上。这样以来，随后的INET套接字调用将会很容易的得到套接字数据结构。在创建套接字时，也建立了指向套接字数据结构的协议操作的指针，这些指针与所使用的协议有关：当使用

TCP时，它们将指向与建立TCP连接有关的一组TCP协议的操作。

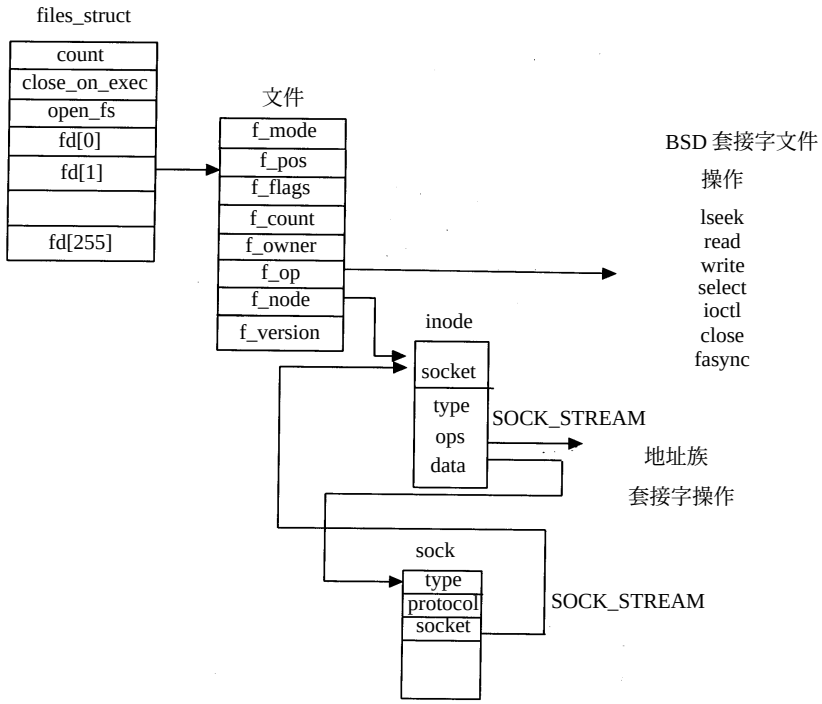

图1-8-5 Linux BSD套接字数据结构

8.4.1 创建BSD套接字

应用程序在使用套接字之前，首先必须拥有一个套接字，系统调用 socket( )向应用程提供创建套接字的手段，该系统调用必须给出所使用的地址族、套接字类型和协议的标志。

首先，系统利用地址族标志找到与之匹配的 pops向量，若地址族较为特殊，则应先由kerneld守护程序载入实现该地址族的功能模块，然后为这一 BSD套接字分配新的套接字数据结构。事实上，套接字数据结构从物理上讲是 VFS(Virtual File System)索引节点数据结构的一部分，分配一个套接字数据结构也就意味着分配一个 VFS索引节点数据结构。不必奇怪，只要想一想套接字也是一种文件就明白了。由于所有的文件都由 VFS索引节点所指代，为了支持文件操作，BSD套接字自然应有一个相应的VFS索引节点。

新创建的BSD套接字数据结构中包含了一个指针，它指向地址族所定义的套接字例程，在proto_ops数据结构中对比进行了设定。该套接字的类型按调用时的要求设定，诸如SOCK_STREAM或SOCK_DGRAM等等。对地址族所定义的创建例程的调用，是通过proto_ops数据结构中保存的地址来进行的。

还要从当前进程的fd(File descriptor)向量中分配一个空闲的文件描述符，并初始化该描述符所指向的file(文件)数据结构，这包括设置文件操作指针指向 BSD套接字接口所支持的一组BSD套接字文件操作。之后的任何文件操作都将直接调用套接字接口，而套接字接口程序将通过调用它的地址族操作例程将其转交给它相应的地址族。

8.4.2 为INET BSD Socket绑定地址

为了侦听从互联网上发来的连接请求，每一个服务器必须先创建一个套接字，然后为其绑定一个地址。通常绑定操作是在INET套接字层利用下层TCP或UDP协议的支持完成的。将要绑定的地址不能正用于其他的连接通信，这意味着套接字的状态必须为 TCP_CLOSE。待绑定地址包括一个IP地址及一个端口号(可选)，通常IP地址已分配给了一个网络设备，这个设备应支持INET地址族并且其接口是活跃而且可用的 (可通过ifconfig命令查看系统中当前活跃的网络接口进程)。IP地址可以为全“ 1 ”或全“ 0 ”的广播地址，这样通信数据将传给每一个网络设备，也可以任意指定一个 IP地址，只要本机是一个透明代理或防火墙，但只有具有超级用户权限的进程才可以这么做。在套接字数据结构中， recv_addr和saddr域保存了绑定的IP地址。若未指明可选的端口号，下层网络将会指定一个可用的端口号。按惯例，小于 1024的端口号对于无超级用户权限的进程是不可用的，因此下层网络通常分配一个大于 1024的端口号。

由于网络设备接收到报文后，还要将其正确地递交到 INET和BSD套接字，因此 TCP和UDP都维护有哈希(hash)地址表，其中保存有IP地址和BSD套接字的映射关系。通过用所收到报文的IP地址检索该表，即可找到相应的套接字，然后正确递交。因为 TCP是一种面向连接协议，所以处理TCP报文要比处理UDP报文使用更多的信息。

UDP哈希表中包括了 sock数据结构指针，通过以端口号作为参数的哈希函数进行检索。由于UDP的哈希表小于实际可用的端口号，所以表中指针通常指向一个 sock数据结构链，它们通过sock中的next指针链接起来。

TCP维护了多个哈希表，因为它相对复杂多了。但注意在操作过程中， TCP并非在进行绑定(bind)操作时将sock数据结构加入到哈希表中，而是在进行侦听 (listen)操作时完成这一加入过程的。绑定操作时， TCP仅仅审查一下所申请端口号是否可用。

8.4.3 建立INET BSD Socket连接

一旦一个套接字创建之后，若其未用于侦听本地进程间连接请求，即可将其用于侦听远程进程间连接请求。对于非面向连接的 UDP，不需作太多的工作，但对于 TCP，由于它是面向连接的，因此需要在两通信进程间建立一条虚电路。

用于建立连接的INET BSD套接字，其状态必须是SS_UNCONNECTED。UDP协议不需建立虚连接，因此它所传输的消息不一定能正确到达，但它支持 BSD套接字的连接操作。一个UDP INET BSD套接字连接操作只是简单地设定好远端进程地址 —— IP地址和端口号，并设置一个路由表入口缓存 (cache)，这样基于本BSD套接字的报文无需再次查询路由数据库 (除非该路由故障)。INET sock数据结构中的ip_route_cache指针指向缓存路由信息，如果这个 BSD套接字未指明地址信息，则使用该路由信息传送消息，并且由 UDP将sock状态置为TCP_ESTABLISHED。

对于TCP BSD套接字连接操作， TCP必须建立一个包含连接信息的消息，并发送到指定的IP地址。这些连接信息包括一个唯一的起始消息序号、初始化主机所能处理的最大消息长度以及接收窗口大小等等。 TCP中所有消息均被编号，首编号用于第一个消息， Linux选用了一种有效的随机方式取到首编号值，以避免恶意的网络攻击。传输过程中，接受方须向发送方进行确认，指示其所收到的正确消息编号，发送方将重传未被确认的消息。传输窗口大小表示在确认了字节之后还可以发送多少个消息。最大消息长度取决于发出初始连接请求的网

络设备，但如果接收方网络设备所支持的最大消息长度更小，则连接将取用两者中较小的。指明传输窗口大小的发送方要等待接收方的接受或拒绝响应，也就是说要等待接收消息，这样就需要将sock加入到tcp_listening_hash中，当所等待消息发来后，即可被正确递交给该 sock 数据结构，同时TCP启动定时器，确定传输是否超时。

8.4.4 INET BSD Socket侦听

套接字绑定了地址后，可能要侦听发给自己的绑定地址的连接请求 (某些应用程序可以不经绑定而直接侦听，这种情况下， INET套接字层会自动为其绑定一个可用的端口号)。侦听套接字程序就将套接字状态置为 TCP_LISTEN，并做好其他一些允许接收连接请求的工作。

对UDP而言，只需设置套接字状态即可；但对于 TCP，还要把套接字的sock数据结构放入两张哈希表中： tcp_bound_hash表和tcp_listonry_hash表，这两张表也都是通过以端口号为参数的哈希函数进行检索的。

一旦一个TCP连接请求到达后， TCP将建立一个新的sock数据结构代表该请求，若连接请求最终被接受，则新的 sock数据结构将成为半个TCP连接，同时复制包含连接请求的 sk_buff，并放入侦听 sock数据结构中的receive_queue队列，复制的sk_buff包含有指向新建 sock数据结构的指数。

8.4.5 接受连接请求

UDP不支持连接概念，因此接受 INET套接字连接请求是对 TCP而言。它引发由原侦听套接字复制新套接字数据结构。接受操作由支持套接字的协议层来完成，也就是由 INET接受发来的连接请求，若下层协议不支持连接 (如UDP)，则INET协议层接收失败；否则，将接受操作递交给TCP协议。接收操作有两种：有阻塞的和无阻塞。无阻塞操作中，若无连接请求到来，接受操作失败，释放新建的套接字数据结构；有阻塞操作中，实现接收操作的网络应用程序会在队列中等待并挂起，直到收到一 TCP连接请求。一旦接收到连接请求，则丢弃包含该请求的sk_buff，将sock数据结构回交给INET套接字层，并在该层将其链接到早先新建的套接字数据结构上。新套接字的文件描述符 (fd)回交给网络应用程序，然后应用程序就可以用文件描述符在新建的INET BSD套接字上进行套接字操作。

8.5 IP层

8.5.1 套接字缓冲区

协议分层为网络传输带来了一些问题，其中之一就是在利用各层发送数据时，每层都要加上自己的头部和尾部信息，而接收时又要由每层将这些信息去掉，这样就使得数据缓冲区的分配变得更为困难，因为每层都要能在缓冲区中找到特定的头部和尾部。一种解决方案是在每一层都对缓冲区进行全拷贝，但这样做效率太低。在 Linux中，各层协议和网络设备驱动程序间只传递套接字缓冲区或 sk_buff，sk_buff中有指针和长度域，这样各层协议即可通过标准函数或方法使用数据。

图1-8-6给出了sk_buff的数据结构，每一个 sk_buff有一个相关联的数据块。 sk_buff中有4个数据指针，用于使用和管理套接字缓冲区中的数据：

- head 指向内存中数据的起始区，一旦分配了 sk_buff及其相关数据块，即可确定该指针。
- data 指向当前协议数据的开始，该指针取决于当前拥有 sk_buff的协议层。
- tail 指向当前协议数据尾，与data一样，它也依赖于当前拥有sk_buff的协议层。
- end 指向内存中数据的结束区，分配了sk_buff后，该值确定。

其中有两个长度域：

a. len 当前协议报文的长度。

b. truesize 整个数据缓冲区长。

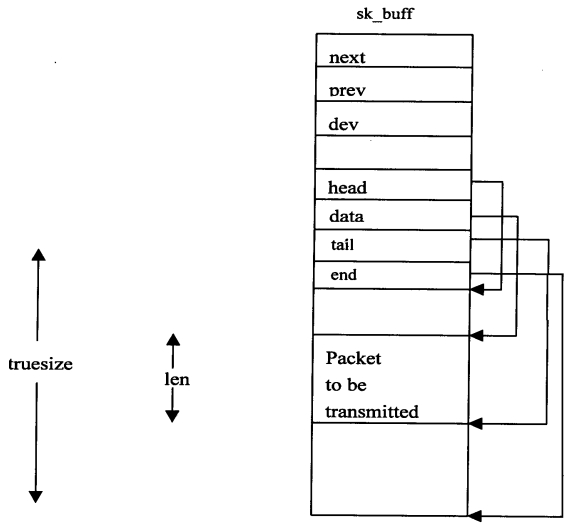

图1-8-6 套接字缓冲区

sk_buff的控制处理代码对添加和删除协议头和尾提供了标准操作，通过这些操作可安全地使用sk_buff中的data、tail和len域：

- push 将data指针指向数据开始区，并相应增加 len域，用于向待传输数据添加数据或协议头。
- pull 将data指针指向数据区的end，并相应减小len域，用于从接收到的数据中删除数据或协议头。
- put 将tail指针指向数据区的end，并相应增加len域，用于向发送的数据end处添加数据或协议信息。
- trim 将tail指针指向数据区起始处，并相应减小 len域，用于从接收到的报文中删除数据或协议尾。

sk_buff数据结构中还有 sk_buff的双向链表指针，并有通用的 sk_buff例程用这些指针从sk_buff链头或链尾对sk_buff数据结构进行添加或删除。

8.5.2 接收IP报文

第6章曾讲述了 Linux是如何将网络驱动程序建立到内核中并进行初始化的，这样在dev_base链中相应就有了一系列的设备数据结构，每个设备数据结构描述了其设备并提供了一组回调例程。当网络各协议层需要网络驱动程序为其工作时，就要调用这些例程，它们通

常和使用网络设备地址的数据传输有关。当网络设备从网上接收到报文后，必须先将接收到的数据转换到sk_buff数据结构中，然后把sk_buff添加到backlog队列中。当backlog队列太长时，新接收到报文的sk_buff将被抛弃，一旦队列中有待处理的sk_buff，网络bottom half即被置为可用状态。

网络bottom half控制处理进程被调度器激活后，先处理待发送报文，然后处理sk_buff的backlog队列，以决定向哪一层递交接收到的报文。Linux初始化网络各层时，每种协议都要注册，它们分别把各自的packet_type数据结构加入到ptype_all或ptype_base表中。packet-type数据结构中包括有协议类型、一个网络设备指针、一个协议接收数据处理例程指针和一个指向下一个packet_type数据结构的指针(用于维护表或哈希链)。ptype_all链用于检测从网络设备接收到的非常用协议报文。ptype_base哈希表以协议标志为索引，用以判别将接收到的网络报文递交给哪个协议。网络bottom half进程对协议类型进行匹配，以免在上述两类表中找到多个入口，但当检测所有网络传输时，的确可能匹配到不止一个入口，这种情况下，将复制sk_buff，所有sk_buff都将递交给匹配到的协议处理例程。

8.5.3 发送IP报文

应用程序相互间交换数据时，要传输报文；网络协议支持建立连接或当连接已经建立好后，也要进行报文传输。无论哪种情况下发送报文，都要首先建立包含数据的sk_buff，并且各层都要对即将发送的数据添加各自的协议头。

sk_buff必须先传送到网络设备然后才能发送。那么首先它要经过协议层，例如IP，由其决定使用哪个网络设备，这取决于发送该报文的最佳路径。如果计算机是通过modem连入网络的，即使用PPP协议，那么路径选择就很简单：通过回送设备发给本地主机或发给PPP modem连接末端的网关；但对于连接到以太网上的计算机而言，则将较为困难，因为网络中有许多台计算机。

对每一个IP报文的发送，IP用路由表解决寻径问题。通过查询路由表返回的rtable数据结构，即可找到到达目的IP地址的正确路径。这种查询要用到源IP地址、网络设备数据结构地址，有时还要用到预编译硬件头，这种硬件头由网络设备定义，内有源和目的物理地址以及其他一些特定的媒体信息。若网络设备为以太网设备，那么它的硬件头如图1-8-7所示，其中的源和目的地址就是以太网物理地址。因为使用任一个路由的IP发送报文都将硬件头加到IP报文之前，而硬件头的构造又需时间，所以将硬件头与IP路由缓存在一起，这样可以提高效率。对硬件头中物理地址的解析可能要用到ARP协议，这时要先把报文存起来，直到地址解析完成后再发送。要缓存解析后的地址信息缓存，之后用到同样接口的IP报文就不用再次进行ARP解析了。

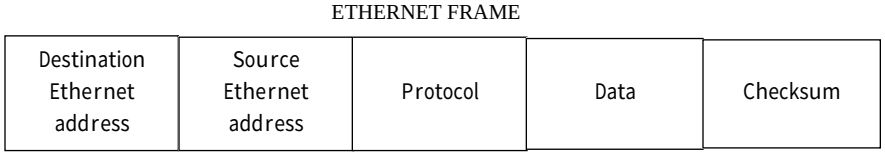

图1-8-7 以太网硬件头

8.5.4 数据分片

每种网络设备都有一个最大报文长度限制，对于大于该长度的报文，它既不能接收，也

无法发送。鉴于此，IP协议能将大的数据分片为几个，以支持网络设备的处理能力。如前所述，IP协议头中的标识域、标志域和片偏移域用于分片与重组工作。

IP报文发送前，先要从IP路由表中找出所用的网络设备。每个设备都有一个最大传输单元(Maximum Transfer Unit, MTU)域，指明该设备所能支持的最大报文长度。若设备的 MTU小于待发送的IP报文长度，则须将IP报文分片，由一个sk_buff代表每一片，片的IP头中的标志域指出其是否分片，片偏移域指出本片在整个 IP报文中的字节偏移量，若在分片过程中无法分配可用的sk_buff，则传输失败。

分片的方法及格式如图1-8-8，该图说明报头长24个字节、数据区长1600个字节的数据报在MTU为700字节的物理网络中分片的情况。

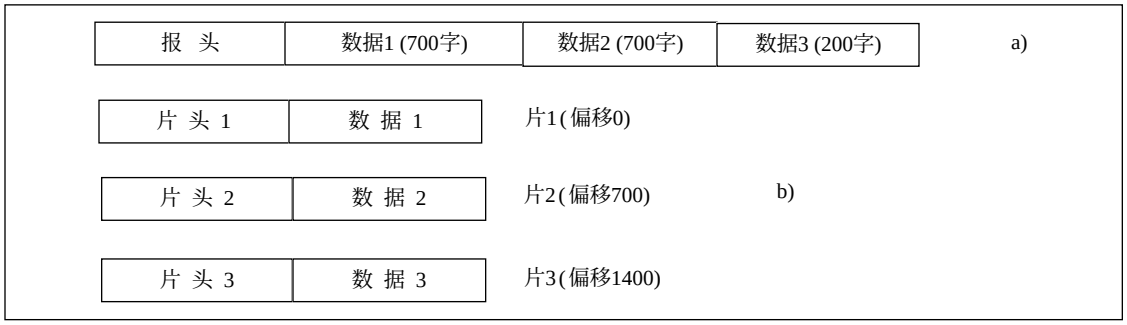

图1-8-8 IP报文的分片与重组

a) 数据区大小为1600字节的初始数据报 b) 在MTU=700字节的网络上的三个分片

接收IP分片会更复杂一些，因为分片可能以各种顺序到达，并且在接收到所有的分片之后才能对其重组。每当接收方收到IP报文后，先检测其是否被分片，当接收到第一个分片时，IP将创建一个新的ipq数据结构，并把它放入等待重组的ipqueue链中。更多的分片到达后，先找到相应的ipq数据结构，并创建一个新的ipfrag数据结构以描述各片。每个ipq数据结构都有唯一的源和目的IP地址、上层协议标识以及IP片标识，所有的分片都到达后，将被组合到一个sk_buff中，然后递交到上一层协议进行处理。每个ipq都有一个定时器，每收到一个正确的分片就重置定时器，若定时器超时，则释放 ipq数据结构以及它的ipfrag数据结构，然后生成一个“丢失”消息，最后将该消息交由上层协议处理。

8.6 地址解析协议

地址解析协议(Address Resolution Protocol, ARP)的任务就是实现IP地址和物理硬件地址(如以太网地址)间的互译，在把数据交由设备驱动程序进行发送之前，IP需要先得到解析结果。对于判别设备是否需要硬件头以及若需要则是否应重建硬件头， IP有多种检测方法，Linux通过缓存硬件头来避免对它们的多次重建。若硬件头确需重建，则调用设备特定的硬件头重建例程，所有的以太网设备使用同一个重建例程，该例程利用 ARP服务将目的IP地址转化为相应的物理地址。

ARP协议本身很简单，仅包括两种消息类型： ARP请求与ARP应答。ARP请求中包含有欲解析的IP地址，应答报文中则包含了对相应IP地址的解析结果 —— 硬件地址。

Linux中的ARP协议层是基于arp_table数据结构表实现的。arp_table数据结构记录了对应于某一IP地址的物理地址解析，当IP地址需要解析时则创建；记录已因过时而失效时则删除，每个arp_table数据结构都包括如下的域：

last used	ARP入口上次访问时间
last updated	ARP入口上次刷新时间
flags	描述接口状态，例如是否完成等
IP address	入口IP地址
hardware address	解析得到的硬件地址
hardware header	缓存的硬件头指针
timer	一个timer_list入口，用于检测ARP是否超时
retries	ARP请求的重试次数
sk_buff queue	sk_buff入口表，表中入口均等待本次ARP结果

ARP表中包含有一个指针表，这些指针指向arp_tables入口链，而这些链均已被缓存，以提高访问速度。以入口的最后两字节的IP地址为索引，对ARP表进行检索，然后在检索到的入口链中顺次查找所要的IP地址，Linux还缓存了hh_cache数据结构，该数据结构中有从arp_table入口中取出的预编译硬件头。

当发出一个IP地址解析请求而又无相应的arp_table入口时，ARP必须发出一个ARP请求消息。ARP先在arp_table入口表中创建一个新的arp_table，并将需要该地址解析结果的sk_buff放入新入口的sk_buff队列中，然后发送ARP请求报文并启动定时器。若超时无响应，将重发一定次数的ARP请求，多次请求仍无响应，则删除新建的arp_table入口，并通知等待的sk_buff数据结构队列，队列中的各sk_buff再请求上层协议处理本次失败操作。对此UDP并不关心是否丢失报文，但TCP将尝试在一条已建立的连接上重发报文，若目的主机响应请求并发回硬件地址，则置该arp_table入口为完成，并从sk_buff队列中移出各项，让它们继续完成发送工作，这时所需硬件地址已写入这些sk_buff的硬件头中。

ARP协议层必须响应指定了IP地址的ARP请求，它先注册协议类型(ETH_P_ARP)，并生成一个packet_type数据结构，这意味着所有由网络设备接收到的ARP报文都将交由它处理，包括ARP请求与响应，它利用保存在接收设备的device数据结构中的硬件地址完成响应。

由于网络拓朴结构可以随时改变，因此IP地址也可以重新分配给不同的硬件地址，例如，一些拨号服务每当建立起一个连接就分配一次IP地址，为了保持ARP表中入口的实时可用，ARP用一个周期性定时器，该定时器将定期检测ARP表以确定超时的入口，但它并不删除含有一个或多个缓存硬件头的入口，这样做会很危险，因为有其他数据结构有赖于这些入口。某些arp_table入口是永久性的，对它们也作有标记以免其被释放。由于每一个arp_table入口都要消耗核态内存，所以ARP表不能太大，一旦要分配一个新入口时发现ARP表已达最大，则通过查找并删除最旧的入口来压缩ARP表。

8.7 IP路由

进行IP寻径时，首先检测路由缓存，若无匹配的路由信息，再搜索转发信息数据库(Forwarding Information Database)，如果仍不成功，则本次IP报文发送失败，并通知应用程序；如果找到了路由信息，就生成一个包含该信息的新入口，并将其加入到路由缓存中。路

由缓存是一张表(ip_rt_hash_table)，表中包含了rtable数据结构链指针，对路由表的检索是通过哈希函数完成的，这个哈希函数以IP地址中最不重要的两个字节为参数，这两个字节应以如下规则选定：对于不同的目的地址应尽可能不相同，这样做可以提供更好的哈希值。每个rtable入口包含了路由信息：目的IP地址、到达目的IP地址所要用到的网络设备、最大消息长度等等。它还有一个引用计数、一个使用计数和一个上次被使用的时间戳。每次用到一个路由，就将其引用记数增1，以标明使用该路由的网络连接数量；应用程序不再使用该路由时，就将引用记数减1。使用计数是在每次对其路由进行了查找时增 1，以此在哈希入口链中对路由入口排序。上次被使用的时间戳用于对路由表周期性地检测，以便找出最旧的信息入口，若某个路由最近未被使用过，则将其删除。路由缓存中的入口以使用次数排序是为了使利用率最高的路由放在哈希队列头，这样可提高路由查找效率。

转发信息数据库

转发信息数据库(见图1-8-9)包含了相对IP当前本系统可用的路由信息，它是一个相当复杂的数据结构。尽管对其已采取了合理而有效的编排处理，但对它的访问仍然很慢，这也正是建立路由缓存的原因：利用已知的可用路由来提高路由查询速度。路由缓存中的信息，均取自转发信息数据库，并且是其中经常被使用的那一部分。

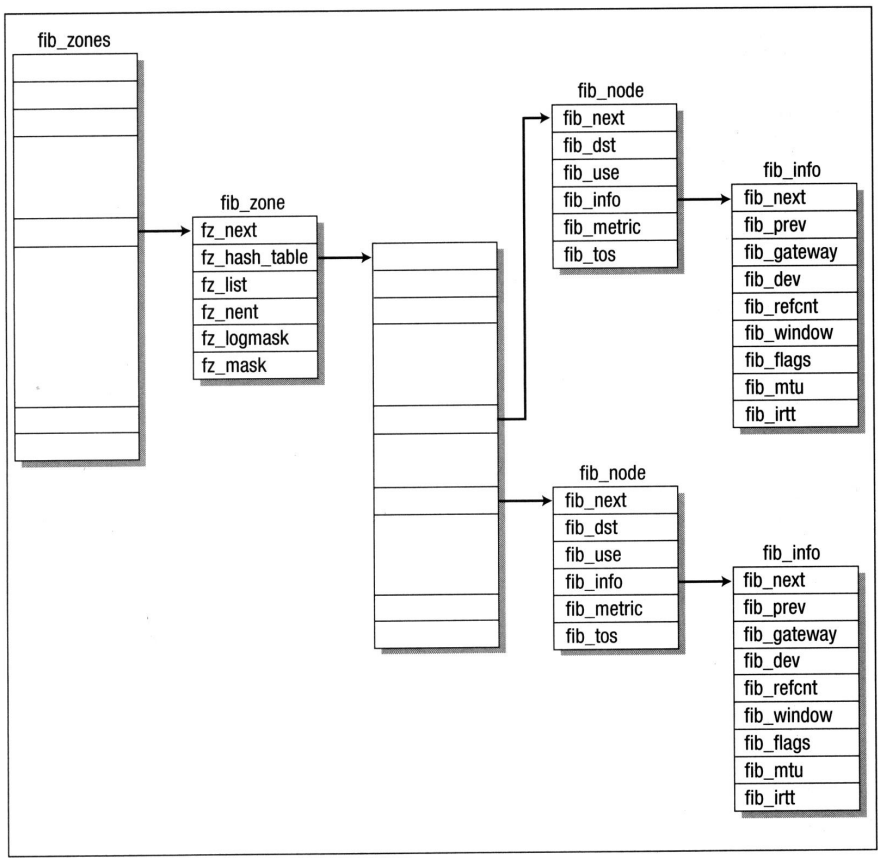

图1-8-9 转发信息数据库

由一个fib_zone数据结构来代表每一个IP子网，fib_zones哈希表中有所有这些fib_zone指针，其索引值取自IP子网掩码。每个fib_zone数据结构中都有fib_list项，所有发往同一子网的路由均由fib_list队列中的fib_node和fib_info数据结构所说明，若一个子网中的路由数过多，系统就会生成一张哈希表，以简化对fib_node数据结构的查找。

对于一个IP子网可能会有多个路由，这些路由将通过不同的几个网关。 IP路由层不允许通往同一子网的多个路由使用同一网关，换句话说，若有通往同一子网的多个路由，对每一个都使用不同的网关进行转发。有一个度量与每个路由相关联，用以指明路由的的优劣度，很重要的一种度量是“跳步(hop)”，它是IP报文到达目的子网前所经过的子网个数，度量值越高，该路由越差。